CS 2257 **OPERATING SYSTEMS LAB** **LTPC**
0 0 3 2

1. Write programs using the following system calls of UNIX operating system:
   fork, exec, getpid, exit, wait, close, stat, opendir, readdir
2. Write programs using the I/O System calls of UNIX operating system (open, read, write, etc).
3. Write C programs to simulate UNIX commands like ls, grep, etc.
4. Given the list of processes, their CPU burst times and arrival times. Display/print the Gantt chart for FCFS and SJF. For each of the scheduling policies, compute and print the average waiting time and average turnaround time.
5. Given the list of processes, their CPU burst times and arrival times. Display/print the Gantt chart for Priority and Round robin. For each of the scheduling policies, compute and print the average waiting time and average turnaround time.
6. Develop application using Inter-Process Communication (using shared memory, pipes or message queues).
7. Implement the Producer-Consumer problem using semaphores (using UNIX system calls)
8. Implement Memory management schemes like paging and segmentation.
9. Implement Memory management schemes like First fit, Best fit and Worst fit.
10. Implement any file allocation techniques (Contiguous, Linked or Indexed).

INDEX

Exp#	Name of the Experiment	Session#
	Process System Calls	
1a	`fork` system call	2
1b	`wait` system call	
1c	`exec` system call	
1d	`stat` system call	
1e	`readdir` system call	
	I/O System Calls	
2a	`creat` system call	1
2b	`read` system call	
2c	`write` system call	
	Command Simulation	
3a	`ls` command	2
3b	`grep` command	
3c	`cp` command	
3d	`rm` command	
	Process Scheduling	
4a	FCFS scheduling	2
4b	SJF scheduling	
4c	Priority scheduling	
4d	Round Robin scheduling	
	Inter-process Communication	
5a	Fibonacci & Prime number	3
5b	`who|wc -l`	
5c	Chat Messaging	
5d	Shared memory	
5e	Producer-Consumer problem	
	Memory Management	
6a	First Fit	2
6b	Best Fit	
6c	FIFO Page Replacement	
6d	LRU Page Replacement	
	File Allocation	
7a	Contiguous	1

PROCESS SYSTEM CALL

`fork()`

- **The fork system call is used to create a new process called *child* process.**
  - **The return value is 0 for a child process.**
  - **The return value is negative if process creation is unsuccessful.**
  - **For the parent process, return value is positive**
- **The child process is an exact copy of the parent process.**
- **Both the child and parent continue to execute the instructions following fork call.**
- **The child can start execution before the parent or vice-versa.**

`getpid()` and `getppid()`

- **The getpid system call returns process ID of the calling process**
- **The getppid system call returns parent process ID of the calling process**

`wait()`

- **The wait system call causes the parent process to be blocked until a child terminates.**
- **When a process terminates, the kernel notifies the parent by sending the SIGCHLD signal to the parent.**
- **Without wait, the parent may finish first leaving a *zombie* child, to be adopted by init process**

`execl()`

- **The exec family of function (execl, execv, execle, execve, execlp, execvp) is used by the child process to load a program and execute.**
- **execl system call requires path, program name and null pointer**

`exit()`

- **The exit system call is used to terminate a process either normally or abnormally**
- **Closes all standard I/O streams.**

`stat()`

- **The stat system call is used to return information about a file as a structure.**

`opendir()`, `readdir()` and `closedir()`

- **The opendir system call is used to open a directory**
  - **It returns a pointer to the first entry**
  - **It returns NULL on error.**
- **The readdir system call is used to read a directory as a *dirent* structure**
  - **It returns a pointer pointing to the next entry in directory stream**
  - **It returns NULL if an error or end-of-file occurs.**
- **The closedir system call is used to close the directory stream**
- **Write to a directory is done only by the kernel.**

Exp# 1a **`fork` system call**

Aim

To create a new child process using fork system call.

Algorithm

1. Declare a variable *x* to be shared by both child and parent.
2. Create a child process using fork system call.
3. If return value is -1 then
    a. Print "Process creation unsuccessfull"
    b. Terminate using exit system call.
4. If return value is 0 then
    a. Print "Child process"
    b. Print process id of the child using getpid system call
    c. Print value of *x*
    d. Print process id of the parent using getppid system call
5. Otherwise
    a. Print "Parent process"
    b. Print process id of the parent using getpid system call
    c. Print value of *x*
    d. Print process id of the shell using getppid system call.
6. Stop

Result

Thus a child process is created with copy of its parent's address space.

Program

```
/* Process creation - fork.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

main()
{
   pid_t  pid;
   int x = 5;
   pid = fork();
   x++;

   if (pid < 0)
   {
      printf("Process creation error");
      exit(-1);
   }
   else if (pid == 0)
   {
      printf("Child process:");
      printf("\nProcess id is %d", getpid());
      printf("\nValue of x is %d", x);
      printf("\nProcess id of parent is %d\n", getppid());
   }
   else
   {
      printf("\nParent process:");
      printf("\nProcess id is %d", getpid());
      printf("\nValue of x is %d", x);
      printf("\nProcess id of shell is %d\n", getppid());
   }
}
```

Output

```
$ gcc fork.c

$ ./a.out
Child process:
Process id is 19499
Value of x is 6
Process id of parent is 19498

Parent process:
Process id is 19498
Value of x is 6
Process id of shell is 3266
```

Exp# 1b `wait` system call

Aim

To block a parent process until child completes using wait system call.

Algorithm

1. Create a child process using fork system call.
2. If return value is -1 then
   a. Print "Process creation unsuccessfull"
3. Terminate using exit system call.
4. If return value is > 0 then
   a. Suspend parent process until child completes using wait system call
   b. Print "Parent starts"
   c. Print even numbers from 0–10
   d. Print "Parent ends"
5. If return value is 0 then
   a. Print "Child starts"
   b. Print odd numbers from 0–10
   c. Print "Child ends"
6. Stop

Result

Thus using wait system call zombie child processes were avoided.

Program

```
/* Wait for child termination - wait.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

main()
{
   int i, status;
   pid_t pid;

   pid = fork();
   if (pid < 0)
   {
      printf("\nProcess creation failure\n");
      exit(-1);
   }
   else if(pid > 0)
   {
      wait(NULL);
      printf ("\nParent starts\nEven Nos: ");
      for (i=2;i<=10;i+=2)
         printf ("%3d",i);
      printf ("\nParent ends\n");
   }
   else if (pid == 0)
   {
      printf ("Child starts\nOdd Nos: ");
      for (i=1;i<10;i+=2)
         printf ("%3d",i);
      printf ("\nChild ends\n");
   }
}
```

Output

```
$ gcc wait.c

$ ./a.out
Child starts
Odd Nos:   1  3  5  7  9
Child ends

Parent starts
Even Nos:   2  4  6  8 10
Parent ends
```

Exp# 1c **exec system call**

Aim

To load an executable program in a child processes exec system call.

Algorithm

1. If no. of command line arguments ≠ 3 then stop.
2. Create a child process using fork system call.
3. If return value is -1 then
   a. Print "Process creation unsuccessfull"
   b. Terminate using exit system call.
4. If return value is > 0 then
   a. Suspend parent process until child completes using wait system call
   b. Print "Child Terminated".
   c. Terminate the parent process.
5. If return value is 0 then
   a. Print "Child starts"
   b. Load the program in the given path into child process using exec system call.
   c. If return value of exec is negative then print the exception and stop.
   d. Terminate the child process.
6. Stop

Result

Thus the child process loads a binary executable file into its address space.

Program

```
/* Load a program in child process - exec.c */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

main(int argc, char*argv[])
{
   pid_t pid;
   int i;

   if (argc != 3)
   {
      printf("\nInsufficient arguments  to load program");
      printf("\nUsage: ./a.out <path> <cmd>\n");
      exit(-1);
   }

   switch(pid = fork())
   {
      case -1:
         printf("Fork failed");
         exit(-1);

      case 0:
         printf("Child process\n");
         i = execl(argv[1], argv[2], 0);
         if (i <  0)
         {
            printf("%s program not loaded using exec system
                    call\n", argv[2]);
            exit(-1);
         }

      default:
         wait(NULL);
         printf("Child Terminated\n");
         exit(0);
   }
}
```

Output

```
$ gcc exec.c

$ ./a.out

Insufficient arguments  to load program
Usage: ./a.out <path> <cmd>

$ ./a.out /bin/ls ls
Child process
a.out               cmdpipe.c           consumer.c
dirlist.c           ex6a.c              ex6b.c
ex6c.c              ex6d.c              exec.c
fappend.c           fcfs.c              fcreate.c
fork.c              fread.c             hello
list                list.c              pri.c
producer.c          rr.c                simls.c
sjf.c               stat.c              wait.c
Child Terminated

$ ./a.out /bin/who who
Child process
who program not loaded using exec system call
Child Terminated

$ ./a.out /usr/bin/who who
Child process
vijai    pts/0        2013-04-24 15:48 (192.168.144.1)
Child Terminated
```

Exp# 1d **`stat` system call**

Aim

To display file status using stat system call.

Algorithm

1. Get *filename* as command line argument.
2. If *filename* does not exist then stop.
3. Call stat system call on the *filename* that returns a structure
4. Display members st_uid, st_gid, st_blksize, st_block, st_size, st_nlink, etc.,
5. Convert time members such as st_atime, st_mtime into time using ctime function
6. Compare st_mode with mode constants such as S_IRUSR, S_IWGRP, S_IXOTH and display file permissions.
7. Stop

Result

Thus attributes of a file is displayed using stat system call.

Program

```
/* File status - stat.c */
#include <stdio.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char*argv[])
{
   struct stat file;
   int n;
   if (argc != 2)
   {
      printf("Usage: ./a.out <filename>\n");
      exit(-1);
   }
   if ((n = stat(argv[1], &file)) == -1)
   {
      perror(argv[1]);
      exit(-1);
   }

   printf("User id : %d\n", file.st_uid);
   printf("Group id : %d\n", file.st_gid);
   printf("Block size : %d\n", file.st_blksize);
   printf("Blocks allocated : %d\n", file.st_blocks);
   printf("Inode no. : %d\n", file.st_ino);
   printf("Last accessed : %s", ctime(&(file.st_atime)));
   printf("Last modified : %s", ctime(&(file.st_mtime)));
   printf("File size : %d bytes\n", file.st_size);
   printf("No. of links : %d\n", file.st_nlink);

   printf("Permissions : ");
   printf( (S_ISDIR(file.st_mode)) ? "d" : "-");
   printf( (file.st_mode & S_IRUSR) ? "r" : "-");
   printf( (file.st_mode & S_IWUSR) ? "w" : "-");
   printf( (file.st_mode & S_IXUSR) ? "x" : "-");
   printf( (file.st_mode & S_IRGRP) ? "r" : "-");
   printf( (file.st_mode & S_IWGRP) ? "w" : "-");
   printf( (file.st_mode & S_IXGRP) ? "x" : "-");
   printf( (file.st_mode & S_IROTH) ? "r" : "-");
   printf( (file.st_mode & S_IWOTH) ? "w" : "-");
   printf( (file.st_mode & S_IXOTH) ? "x" : "-");
   printf("\n");

   if(file.st_mode & S_IFREG)
      printf("File type : Regular\n");
   if(file.st_mode & S_IFDIR)
      printf("File type : Directory\n");
}
```

Output

```
$ gcc stat.c

$ ./a.out fork.c
User id : 0
Group id : 0
Block size : 4096
Blocks allocated : 8
Inode no. : 16627
Last accessed : Fri Feb 22 21:57:09 2013
Last modified : Fri Feb 22 21:56:13 2013
File size : 591 bytes
No. of links : 1
Permissions : -rw-r--r--
File type : Regular
```

Exp# 1e **`readdir` system call**

Aim

To display directory contents using readdir system call.

Algorithm

1. Get directory *name* as command line argument.
2. If directory does not exist then stop.
3. Open the directory using opendir system call that returns a structure
4. Read the directory using readdir system call that returns a structure
5. Display d_name member for each entry.
6. Close the directory using closedir system call.
7. Stop

Result

Thus files and subdirectories in the directory was listed that includes hidden files.

Program

```
/* Directory content listing - dirlist.c */

#include <stdio.h>
#include <dirent.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
   struct dirent *dptr;
   DIR *dname;

   if (argc != 2)
   {
      printf("Usage: ./a.out <dirname>\n");
      exit(-1);
   }

   if((dname = opendir(argv[1])) == NULL)
   {
      perror(argv[1]);
      exit(-1);
   }

   while(dptr=readdir(dname))
      printf("%s\n", dptr->d_name);

   closedir(dname);
}
```

Output

```
$ gcc dirlist.c

$ ./a.out vijai
wait.c
a.out
..
stat.c
dirlist.c
fork.c
.
exec.c
```

FILE SYSTEM CALL

open()

- **Used to open an existing file for reading/writing or to create a new file.**
- **Returns a file descriptor whose value is negative on error.**
- **The mandatory flags are O_RDONLY, O_WRONLY and O_RDWR**
- **Optional flags include O_APPEND, O_CREAT, O_TRUNC, etc**
- **The flags are ORed.**
- **The mode specifies permissions for the file.**

creat()

- **Used to create a new file and open it for writing.**
- **It is replaced with open() with flags O_WRONLY|O_CREAT | O_TRUNC**

read()

- **Reads no. of bytes from the file or from the terminal.**
- **If read is successful, it returns no. of bytes read.**
- **The file offset is incremented by no. of bytes read.**
- **If end-of-file is encountered, it returns 0.**

write()

- **Writes no. of bytes onto the file.**
- **After a successful write, file's offset is incremented by the no. of bytes written.**
- **If any error due to insufficient storage space, write fails.**

close()

- **Closes a opened file.**
- **When process terminates, files associated with the process are automatically closed.**

Exp# 2a **open system call**

Aim

To create a file and to write contents.

Algorithm

1. Declare a character buffer *buf* to store 100 bytes.
2. Get the new filename as command line argument.
3. Create a file with the given name using open system call with O_CREAT and O_TRUNC options.
4. Check the file descriptor.
   a) If file creation is unsuccessful, then stop.
5. Get input from the console until user types Ctrl+D
   a) Read 100 bytes (max.) from console and store onto buf using read system call
   b) Write length of *buf* onto file using write system call.
6. Close the file using close system call.
7. Stop

Result

Thus a file has been created with input from the user. The process can be verified by using cat command.

Program

```
/* File creation - fcreate.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

main(int argc, char *argv[])
{
   int fd, n, len;
   char buf[100];

   if (argc != 2)
   {
      printf("Usage: ./a.out <filename>\n");
      exit(-1);
   }

   fd = open(argv[1], O_WRONLY|O_CREAT|O_TRUNC, 0644);
   if(fd < 0)
   {
      printf("File creation problem\n");
      exit(-1);
   }

   printf("Press Ctrl+D at end in a new line:\n");
   while((n = read(0, buf, sizeof(buf))) > 0)
   {
      len = strlen(buf);
      write(fd, buf, len);
   }
   close(fd);
}
```

Output

```
$ gcc fcreate.c

$ ./a.out hello
File I/O
Open system call is used to either open or create a file.
creat system call is used to create a file. It is seldom used.
^D
```

Exp# 2b **`read` system call**

Aim

To read the given file and to display file contents.

Algorithm

1. Declare a character buffer *buf* to store 100 bytes.
2. Get existing filename as command line argument.
3. Open the file for reading using open system call with O_RDONLY option.
4. Check the file descriptor.
   a) If file does not exist, then stop.
5. Read until end-of-file using read system call.
   a) Read 100 bytes (max.) from file and print it
6. Close the file using close system call.
7. Stop

Result

Thus the given file is read and displayed on the console. The process can be verified by using cat command.

Program

```
/* File Read  - fread.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

main(int argc, char *argv[])
{
   int fd,i;
   char buf[100];
   if (argc < 2)
   {
      printf("Usage: ./a.out <filename>\n");
      exit(-1);
   }

   fd = open(argv[1], O_RDONLY);
   if(fd == -1)
   {
      printf("%s file does not exist\n", argv[1]);
      exit(-1);
   }

   printf("Contents of the file %s is : \n", argv[1]);
   while(read(fd, buf, sizeof(buf)) > 0)
      printf("%s", buf);

   close(fd);
}
```

Output

```
$ gcc fread.c

$ ./a.out hello
File I/O
open system call is used to either open or create a file.
creat system call is used to create a file. It is seldom used.
```

Exp# 2c **`write` system call**

Aim

To append content to an existing file.

Algorithm

1. Declare a character buffer *buf* to store 100 bytes.
2. Get exisiting filename as command line argument.
3. Create a file with the given name using `open` system call with `O_APPEND` option.
4. Check the file descriptor.
   a) If value is negative, then stop.
5. Get input from the console until user types Ctrl+D
   a) Read 100 bytes (max.) from console and store onto buf using `read` system call
   b) Write length of *buf* onto file using `write` system call.
6. Close the file using close system call.
7. Stop

Result

Thus contents have been written to end of the file. The process can be verified by using `cat` command.

Program

```
/* File append - fappend.c */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>

main(int argc, char *argv[])
{
   int fd, n, len;
   char buf[100];

   if (argc != 2)
   {
      printf("Usage: ./a.out <filename>\n");
      exit(-1);
   }

   fd = open(argv[1], O_APPEND|O_WRONLY|O_CREAT, 0644);
   if (fd < 0)
   {
      perror(argv[1]);
      exit(-1);
   }

   while((n = read(0, buf, sizeof(buf))) > 0)
   {
      len = strlen(buf);
      write(fd, buf, len);
   }

   close(fd);
}
```

Output

```
$ gcc fappend.c

$ ./a.out hello
read system call is used to read from file or console
write system call is used to write to file.
^D
```

COMMAND SIMULATION

- **Using UNIX system calls, most commands can be emulated in a similar manner.**
- **Simulating a command and all of its options is an exhaustive exercise.**
- **Command simulation harnesses one's programming skills.**
- **Command simulation helps in development of standard routines to be customized to the application needs.**
- **Generally file I/O commands are simulated.**

Exp# 3a **`ls` command**

Aim

To simulate ls command using UNIX system calls.

Algorithm

1. Store path of current working directory using getcwd system call.
2. Scan directory of the stored path using scandir system call and sort the resultant array of structure.
3. Display dname member for all entries if it is not a hidden file.
4. Stop.

Result

Thus the filenames/subdirectories are listed, similar to ls command.

Program

```
/* ls command simulation - list.c */

#include <stdio.h>
#include <dirent.h>

main()
{
   struct dirent **namelist;
   int n,i;
   char pathname[100];

   getcwd(pathname);
   n = scandir(pathname, &namelist, 0, alphasort);

   if(n < 0)
      printf("Error\n");
   else
      for(i=0; i<n; i++)
         if(namelist[i]->d_name[0] != '.')
            printf("%-20s", namelist[i]->d_name);
}
```

Output

```
$ gcc list.c -o list

$ ./list
a.out            cmdpipe.c        consumer.c
dirlist.c        ex6a.c           ex6b.c
ex6c.c           ex6d.c           exec.c
fappend.c        fcfs.c           fcreate.c
fork.c           fread.c          hello
list             list.c           pri.c
producer.c       rr.c             simls.c
sjf.c            stat.c           wait.c
```

Exp# 3b **`grep` command**

Aim

To simulate `grep` command using UNIX system call.

Algorithm

1. Get filename and search string as command-line argument.
2. Open the file in read-only mode using `open` system call.
3. If file does not exist, then stop.
4. Let length of the search string be *n*.
5. Read line-by-line until end-of-file
    a. Check to find out the occurrence of the search string in a line by examining characters in the range 1–n, 2–n+1, etc.
    b. If search string exists, then print the line.
6. Close the file using `close` system call.
7. Stop.

Result

Thus the program simulates `grep` command by listing lines containing the search text.

Program

```
/* grep command simulation - mygrep.c */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

main(int argc,char *argv[])
{
   FILE *fd;
   char str[100];
   char c;
   int i, flag, j, m, k;
   char temp[30];
   if(argc != 3)
   {
      printf("Usage: gcc mygrep.c –o mygrep\n");
      printf("Usage: ./mygrep <search_text> <filename>\n");
      exit(-1);
   }

   fd = fopen(argv[2],"r");
   if(fd == NULL)
   {
      printf("%s is not exist\n",argv[2]);
      exit(-1);
   }

   while(!feof(fd))
   {
      i = 0;
      while(1)
      {
         c = fgetc(fd);
         if(feof(fd))
         {
            str[i++] = '\0';
            break;
         }
         if(c == '\n')
         {
            str[i++] = '\0';
            break;
         }
         str[i++] = c;
      }
```

```
      if(strlen(str) >= strlen(argv[1]))
         for(k=0; k<=strlen(str)-strlen(argv[1]); k++)
         {
            for(m=0; m<strlen(argv[1]); m++)
               temp[m] = str[k+m];
            temp[m] = '\0';
            if(strcmp(temp,argv[1]) == 0)
            {
               printf("%s\n",str);
               break;
            }
         }
   }
}
```

Output

```
$ gcc mygrep.c -o mygrep

$ ./mygrep printf dirlist.c
                printf("Usage: ./a.out <dirname>\n");
                printf("%s\n", dptr->d_name);
```

Exp# 3c **cp command**

Aim

To simulate cp command using UNIX system call.

Algorithm

1. Get source and destination *filename* as command-line argument.
2. Declare a buffer of size 1KB
3. Open the source file in readonly mode using open system call.
4. If file does not exist, then stop.
5. Create the destination file using creat system call.
6. If file cannot be created, then stop.
7. File copy is achieved as follows:
    a. Read 1KB data from source file and store onto buffer using read system call.
    b. Write the buffer contents onto destination file using write system call.
    c. If end-of-file then step 8 else step 7a.
8. Close source and destination file using close system call.
9. Stop.

Result

Thus a file is copied using file I/O. The cmp command can be used to verify that contents of both file are same

Program

```
/* cp command simulation - copy.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>

#define SIZE 1024

main(int argc, char *argv[])
{
   int src, dst, nread;
   char buf[SIZE];

   if (argc != 3)
   {
      printf("Usage: gcc copy.c -o copy\n");
      printf("Usage: ./copy <filename> <newfile> \n");
      exit(-1);
   }

   if ((src = open(argv[1], O_RDONLY)) == -1)
   {
      perror(argv[1]);
      exit(-1);
   }

   if ((dst = creat(argv[2], 0644)) == -1)
   {
      perror(argv[1]);
      exit(-1);
   }

   while ((nread = read(src, buf, SIZE)) > 0)
   {
      if (write(dst, buf, nread) == -1)
      {
             printf("can't write\n");
             exit(-1);
      }
   }

   close(src);
   close(dst);
}
```

Output

```
$ gcc copy.c -o copy
$ ./copy hello hello.txt
```

Exp# 3d rm command

Aim

To simulate rm command using UNIX system call.

Algorithm

1. Get *filename* as command-line argument.
2. Open the file in read-only mode using read system call.
3. If file does not exist, then stop.
4. Close the file using close system call.
5. Delete the file using unlink system call.
6. Stop.

Result

Thus files can be deleted in a manner similar to rm command. The deletion of file can be verified by using ls command.

Program

```
/* rm command simulation - del.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

main(int argc, char* argv[])
{
   int fd;

   if (argc != 2)
   {
      printf("Usage: gcc del.c -o del\n");
      printf("Usage: ./del <filename>\n");
      exit(-1);
   }

   fd = open(argv[1], O_RDONLY);
   if (fd != -1)
   {
      close(fd);
      unlink(argv[1]);
   }
   else
      perror(argv[1]);
}
```

Output

```
$ gcc del.c -o del
$ ./del hello.txt
```

PROCESS SCHEDULING

- **CPU scheduling is used in multiprogrammed operating systems.**
- **By switching CPU among processes, efficiency of the system can be improved.**
- **Some scheduling algorithms are FCFS, SJF, Priority, Round-Robin, etc.**
- **Gantt chart provides a way of visualizing CPU scheduling and enables to understand better.**

First Come First Serve (FCFS)

- **Process that comes first is processed first**
- **FCFS scheduling is non-preemptive**
- **Not efficient as it results in long average waiting time.**
- **Can result in starvation, if processes at beginning of the queue have long bursts.**

Shortest Job First (SJF)

- **Process that requires smallest burst time is processed first.**
- **SJF can be preemptive or non–preemptive**
- **When two processes require same amount of CPU utilization, FCFS is used to break the tie.**
- **Generally efficient as it results in minimal average waiting time.**
- **Can result in starvation, since long critical processes may not be processed.**

Priority

- **Process that has higher priority is processed first.**
- **Prioirty can be preemptive or non–preemptive**
- **When two processes have same priority, FCFS is used to break the tie.**
- **Can result in starvation, since low priority processes may not be processed.**

Round Robin

- **All processes are processed one by one as they have arrived, but in rounds.**
- **Each process cannot take more than the time slice per round.**
- **Round robin is a fair preemptive scheduling algorithm.**
- **A process that is yet to complete in a round is preempted after the time slice and put at the end of the queue.**
- **When a process is completely processed, it is removed from the queue.**

Exp# 4a FCFS Scheduling

Aim

To schedule snapshot of processes queued according to FCFS (First Come First Serve) scheduling.

Algorithm

1. Define an array of structure *process* with members *pid*, *btime*, *wtime* & *ttime*.
2. Get length of the ready queue, i.e., number of process (say *n*)
3. Obtain *btime* for each process.
4. The *wtime* for first process is 0.
5. Compute *wtime* and *ttime* for each process as:
   a. $wtime_{i+1} = wtime_i + btime_i$
   b. $ttime_i = wtime_i + btime_i$
6. Compute average waiting time *awat* and average turnaround time *atur*
7. Display the *btime*, *ttime* and *wtime* for each process.
8. Display GANTT chart for the above scheduling
9. Display *awat* time and *atur*
10. Stop

Result

Thus waiting time & turnaround time for processes based on FCFS scheduling was computed and the average waiting time was determined.

Program

```
/* FCFS Scheduling  - fcfs.c */

#include <stdio.h>

struct process
{
   int pid;
   int btime;
   int wtime;
   int ttime;
} p[10];

main()
{
   int i,j,k,n,ttur,twat;
   float awat,atur;

   printf("Enter no. of process : ");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
      printf("Burst time for process P%d (in ms) : ",(i+1));
      scanf("%d", &p[i].btime);
      p[i].pid = i+1;
   }

   p[0].wtime = 0;
   for(i=0; i<n; i++)
   {
      p[i+1].wtime = p[i].wtime + p[i].btime;
      p[i].ttime = p[i].wtime + p[i].btime;
   }
   ttur = twat = 0;
   for(i=0; i<n; i++)
   {
      ttur += p[i].ttime;
      twat += p[i].wtime;
   }
   awat = (float)twat / n;
   atur = (float)ttur / n;
```

```
   printf("\n       FCFS Scheduling\n\n");
   for(i=0; i<28; i++)
      printf("-");
   printf("\nProcess B-Time T-Time W-Time\n");
   for(i=0; i<28; i++)
      printf("-");
   for(i=0; i<n; i++)
      printf("\n  P%d\t%4d\t%3d\t%2d",
               p[i].pid,p[i].btime,p[i].ttime,p[i].wtime);
   printf("\n");
   for(i=0; i<28; i++)
      printf("-");

   printf("\n\nGANTT Chart\n");
   printf("-");
   for(i=0; i<(p[n-1].ttime + 2*n); i++)
      printf("-");
   printf("\n");
   printf("|");
   for(i=0; i<n; i++)
   {
      k = p[i].btime/2;
      for(j=0; j<k; j++)
         printf(" ");
      printf("P%d",p[i].pid);
      for(j=k+1; j<p[i].btime; j++)
         printf(" ");
      printf("|");
   }
   printf("\n");
   printf("-");
   for(i=0; i<(p[n-1].ttime + 2*n); i++)
      printf("-");
   printf("\n");
   printf("0");
   for(i=0; i<n; i++)
   {
      for(j=0; j<p[i].btime; j++)
         printf(" ");
      printf("%2d",p[i].ttime);
   }

   printf("\n\nAverage waiting time     : %5.2fms", awat);
   printf("\nAverage turn around time : %5.2fms\n", atur);
}
```

Output

```
$ gcc fcfs.c

$./a.out
Enter no. of process : 4
Burst time for process P1 (in ms) : 10
Burst time for process P2 (in ms) : 4
Burst time for process P3 (in ms) : 11
Burst time for process P4 (in ms) : 6

       FCFS Scheduling

----------------------------
Process B-Time T-Time W-Time
----------------------------
  P1      10     10      0
  P2       4     14     10
  P3      11     25     14
  P4       6     31     25
----------------------------

GANTT Chart
----------------------------------------
|     P1    |  P2 |     P3      |   P4  |
----------------------------------------
0          10    14            25      31

Average waiting time     : 12.25ms
Average turn around time : 20.00ms
```

Exp# 4b **SJF Scheduling**

Aim

To schedule snapshot of processes queued according to SJF (Shortest Job First) scheduling.

Algorithm

1. Define an array of structure *process* with members *pid*, *btime*, *wtime* & *ttime*.
2. Get length of the ready queue, i.e., number of process (say *n*)
3. Obtain *btime* for each process.
4. *Sort* the processes according to their *btime* in ascending order.
   a. If two process have same *btime*, then FCFS is used to resolve the tie.
5. The *wtime* for first process is 0.
6. Compute *wtime* and *ttime* for each process as:
   a. $wtime_{i+1} = wtime_i + btime_i$
   b. $ttime_i = wtime_i + btime_i$
7. Compute average waiting time *awat* and average turn around time *atur*.
8. Display *btime*, *ttime* and *wtime* for each process.
9. Display GANTT chart for the above scheduling
10. Display *awat* and *atur*
11. Stop

Result

Thus waiting time & turnaround time for processes based on SJF scheduling was computed and the average waiting time was determined.

Program

```
/* SJF Scheduling – sjf.c */

#include <stdio.h>

struct process
{
   int pid;
   int btime;
   int wtime;
   int ttime;
} p[10], temp;

main()
{
   int i,j,k,n,ttur,twat;
   float await,atur;

   printf("Enter no. of process : ");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
      printf("Burst time for process P%d (in ms) : ",(i+1));
      scanf("%d", &p[i].btime);
      p[i].pid = i+1;
   }

   for(i=0; i<n-1; i++)
   {
      for(j=i+1; j<n; j++)
      {
         if((p[i].btime > p[j].btime) ||
            (p[i].btime == p[j].btime && p[i].pid > p[j].pid))
         {
            temp = p[i];
            p[i] = p[j];
            p[j] = temp;
         }

      }
   }
   p[0].wtime = 0;
   for(i=0; i<n; i++)
   {
      p[i+1].wtime = p[i].wtime + p[i].btime;
      p[i].ttime = p[i].wtime + p[i].btime;
   }
   ttur = twat = 0;
```

```
   for(i=0; i<n; i++)
   {
      ttur += p[i].ttime;
      twat += p[i].wtime;
   }
   awat = (float)twat / n;
   atur = (float)ttur / n;

   printf("\n       SJF Scheduling\n\n");
   for(i=0; i<28; i++)
      printf("-");
   printf("\nProcess B-Time T-Time W-Time\n");
   for(i=0; i<28; i++)
      printf("-");
   for(i=0; i<n; i++)
      printf("\n  P%-4d\t%4d\t%3d\t%2d",
               p[i].pid,p[i].btime,p[i].ttime,p[i].wtime);
   printf("\n");
   for(i=0; i<28; i++)
      printf("-");

   printf("\n\nGANTT Chart\n");
   printf("-");
   for(i=0; i<(p[n-1].ttime + 2*n); i++)
      printf("-");
   printf("\n|");
   for(i=0; i<n; i++)
   {
      k = p[i].btime/2;
      for(j=0; j<k; j++)
         printf(" ");
      printf("P%d",p[i].pid);
      for(j=k+1; j<p[i].btime; j++)
         printf(" ");
      printf("|");
   }
   printf("\n-");
   for(i=0; i<(p[n-1].ttime + 2*n); i++)
      printf("-");
   printf("\n0");
   for(i=0; i<n; i++)
   {
      for(j=0; j<p[i].btime; j++)
         printf(" ");
      printf("%2d",p[i].ttime);
   }

   printf("\n\nAverage waiting time     : %5.2fms", awat);
   printf("\nAverage turn around time : %5.2fms\n", atur);
}
```

Output

```
$ gcc sjf.c

$./a.out
Enter no. of process : 5
Burst time for process P1 (in ms) : 10
Burst time for process P2 (in ms) : 6
Burst time for process P3 (in ms) : 5
Burst time for process P4 (in ms) : 6
Burst time for process P5 (in ms) : 9

       SJF Scheduling

----------------------------
Process B-Time T-Time W-Time
----------------------------
  P3       5      5      0
  P2       6     11      5
  P4       6     17     11
  P5       9     26     17
  P1      10     36     26
----------------------------

GANTT Chart
-----------------------------------------------
|  P3  |   P2   |   P4  |     P5     |     P1     |
-----------------------------------------------
0      5      11      17          26          36

Average waiting time     : 11.80ms
Average turn around time : 19.00ms
```

Exp# 4c **Priority Scheduling**

Aim

To schedule snapshot of processes queued according to Priority scheduling.

Algorithm

1. Define an array of structure *process* with members *pid*, *btime*, *pri*, *wtime* & *ttime*.
2. Get length of the ready queue, i.e., number of process (say *n*)
3. Obtain *btime* and *pri* for each process.
4. *Sort* the processes according to their *pri* in ascending order.
   a. If two process have same *pri*, then FCFS is used to resolve the tie.
5. The *wtime* for first process is 0.
6. Compute *wtime* and *ttime* for each process as:
   a. $wtime_{i+1} = wtime_i + btime_i$
   b. $ttime_i = wtime_i + btime_i$
7. Compute average waiting time *awat* and average turn around time *atur*
8. Display the *btime*, *pri*, *ttime* and *wtime* for each process.
9. Display GANTT chart for the above scheduling
10. Display *awat* and *atur*
11. Stop

Result

Thus waiting time & turnaround time for processes based on Priority scheduling was computed and the average waiting time was determined.

Program

```
/* Priority Scheduling  - pri.c */

#include <stdio.h>

struct process
{
   int pid;
   int btime;
   int pri;
   int wtime;
   int ttime;
} p[10], temp;

main()
{
   int i,j,k,n,ttur,twat;
   float awat,atur;

   printf("Enter no. of process : ");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
      printf("Burst time for process P%d (in ms) : ", (i+1));
      scanf("%d", &p[i].btime);
      printf("Priority for process P%d : ", (i+1));
      scanf("%d", &p[i].pri);
      p[i].pid = i+1;
   }

   for(i=0; i<n-1; i++)
   {
      for(j=i+1; j<n; j++)
      {
         if((p[i].pri > p[j].pri) ||
            (p[i].pri == p[j].pri && p[i].pid > p[j].pid) )
         {
            temp = p[i];
            p[i] = p[j];
            p[j] = temp;
         }

      }
   }
   p[0].wtime = 0;
   for(i=0; i<n; i++)
   {
      p[i+1].wtime = p[i].wtime + p[i].btime;
      p[i].ttime = p[i].wtime + p[i].btime;
   }
```

```
   ttur = twat = 0;
   for(i=0; i<n; i++)
   {
      ttur += p[i].ttime;
      twat += p[i].wtime;
   }
   awat = (float)twat / n;
   atur = (float)ttur / n;

   printf("\n\t Priority Scheduling\n\n");
   for(i=0; i<38; i++)
      printf("-");
   printf("\nProcess B-Time Priority T-Time  W-Time\n");
   for(i=0; i<38; i++)
      printf("-");
   for (i=0; i<n; i++)
      printf("\n  P%-4d\t%4d\t%3d\t%4d\t%4d",
         p[i].pid,p[i].btime,p[i].pri,p[i].ttime,p[i].wtime);
   printf("\n");
   for(i=0; i<38; i++)
      printf("-");

   printf("\n\nGANTT Chart\n");
   printf("-");
   for(i=0; i<(p[n-1].ttime + 2*n); i++)
      printf("-");
   printf("\n|");
   for(i=0; i<n; i++)
   {
      k = p[i].btime/2;
      for(j=0; j<k; j++)
         printf(" ");
      printf("P%d",p[i].pid);
      for(j=k+1; j<p[i].btime; j++)
         printf(" ");
      printf("|");
   }
   printf("\n-");
   for(i=0; i<(p[n-1].ttime + 2*n); i++)
      printf("-");
   printf("\n0");
   for(i=0; i<n; i++)
   {
      for(j=0; j<p[i].btime; j++)
         printf(" ");
      printf("%2d",p[i].ttime);
   }

   printf("\n\nAverage waiting time     : %5.2fms", awat);
   printf("\nAverage turn around time : %5.2fms\n", atur);
}
```

Output

```
$ gcc pri.c

$ ./a.out
Enter no. of process : 5
Burst time for process P1 (in ms) : 10
Priority for process P1 : 3
Burst time for process P2 (in ms) : 7
Priority for process P2 : 1
Burst time for process P3 (in ms) : 6
Priority for process P3 : 3
Burst time for process P4 (in ms) : 13
Priority for process P4 : 4
Burst time for process P5 (in ms) : 5
Priority for process P5 : 2

         Priority Scheduling

--------------------------------------
Process B-Time Priority T-Time  W-Time
--------------------------------------
  P2       7       1        7        0
  P5       5       2       12        7
  P1      10       3       22       12
  P3       6       3       28       22
  P4      13       4       41       28
--------------------------------------

GANTT Chart
----------------------------------------------------
|   P2    |  P5  |     P1     |   P3  |      P4      |
----------------------------------------------------
0        7     12           22      28             41

Average waiting time     : 13.80ms
Average turn around time : 22.00ms
```

Exp# 4d **Round Robin Scheduling**

Aim

To schedule snapshot of processes queued according to Round robin scheduling.

Algorithm

1. Get length of the ready queue, i.e., number of process (say n)
2. Obtain *Burst* time B_i for each processes P_i.
3. Get the *time slice* per round, say TS
4. Determine the number of rounds for each process.
5. The wait time for first process is 0.
6. If $B_i > TS$ then process takes more than one round. Therefore turnaround and waiting time should include the time spent for other remaining processes in the same round.
7. Calculate *average* waiting time and turn around time
8. Display the GANTT chart that includes
   a. order in which the processes were processed in progression of rounds
   b. Turnaround time T_i for each process in progression of rounds.
9. Display the *burst* time, *turnaround* time and *wait* time for each process (in order of rounds they were processed).
10. Display *average* wait time and turnaround time
11. Stop

Result

Thus waiting time and turnaround time for processes based on Round robin scheduling was computed and the average waiting time was determined.

Program

```
/* Round robin scheduling  - rr.c */

#include <stdio.h>

main()
{
   int i,x=-1,k[10],m=0,n,t,s=0;
   int a[50],temp,b[50],p[10],bur[10],bur1[10];
   int wat[10],tur[10],ttur=0,twat=0,j=0;
   float awat,atur;

   printf("Enter no. of process : ");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
      printf("Burst time for process P%d : ", (i+1));
      scanf("%d", &bur[i]);
      bur1[i] = bur[i];
   }
   printf("Enter the time slice (in ms) : ");
   scanf("%d", &t);

   for(i=0; i<n; i++)
   {
      b[i] = bur[i] / t;
      if((bur[i]%t) != 0)
         b[i] += 1;
      m += b[i];
   }

   printf("\n\t\tRound Robin Scheduling\n");

   printf("\nGANTT Chart\n");
   for(i=0; i<m; i++)
      printf("--------");
   printf("\n");

   a[0] = 0;
   while(j < m)
   {
      if(x == n-1)
         x = 0;
      else
         x++;
      if(bur[x] >= t)
      {
         bur[x] -= t;
         a[j+1] = a[j] + t;
```

```
      if(b[x] == 1)
      {
         p[s] = x;
         k[s] = a[j+1];
         s++;
      }
      j++;
      b[x] -= 1;
      printf("  P%d    |", x+1);
   }
   else if(bur[x] != 0)
   {
      a[j+1] = a[j] + bur[x];
      bur[x] = 0;
      if(b[x] == 1)
      {
         p[s] = x;
         k[s] = a[j+1];
         s++;
      }
      j++;
      b[x] -= 1;
      printf("  P%d  |",x+1);
   }
}

printf("\n");
for(i=0;i<m;i++)
   printf("--------");
printf("\n");

for(j=0; j<=m; j++)
   printf("%d\t", a[j]);

for(i=0; i<n; i++)
{
   for(j=i+1; j<n; j++)
   {
      if(p[i] > p[j])
      {
         temp = p[i];
         p[i] = p[j];
         p[j] = temp;

         temp = k[i];
         k[i] = k[j];
         k[j] = temp;
      }
   }
}
```

```
   for(i=0; i<n; i++)
   {
      wat[i] = k[i] - bur1[i];
      tur[i] = k[i];
   }
   for(i=0; i<n; i++)
   {
      ttur += tur[i];
      twat += wat[i];
   }

   printf("\n\n");
   for(i=0; i<30; i++)
      printf("-");
   printf("\nProcess\tBurst\tTrnd\tWait\n");
   for(i=0; i<30; i++)
      printf("-");
   for (i=0; i<n; i++)
      printf("\nP%-4d\t%4d\t%4d\t%4d", p[i]+1, bur1[i],
                    tur[i],wat[i]);
   printf("\n");
   for(i=0; i<30; i++)
      printf("-");

   awat = (float)twat / n;
   atur = (float)ttur / n;
   printf("\n\nAverage waiting time     : %.2f ms", awat);
   printf("\nAverage turn around time : %.2f ms\n", atur);
}
```

Output

```
$ gcc rr.c

$ ./a.out
Enter no. of process : 5
Burst time for process P1 : 10
Burst time for process P2 : 29
Burst time for process P3 : 3
Burst time for process P4 : 7
Burst time for process P5 : 12
Enter the time slice (in ms) : 10

                Round Robin Scheduling

GANTT Chart
--------------------------------------------------------------
P1    |  P2    |  P3  |  P4  |  P5  |  P2    |  P5    |    P2  |
--------------------------------------------------------------
0    10       20     23      30      40      50       52      61

------------------------------
Process Burst   Trnd    Wait
------------------------------
P1        10      10       0
P2        29      61      32
P3         3      23      20
P4         7      30      23
P5        12      52      40
------------------------------

Average waiting time     : 23.00 ms
Average turn around time : 35.20 ms
```

INTERPROCESS COMMUNICATION

- **Inter-Process communication (IPC), is the mechanism whereby one process can communicate with another process, i.e exchange data.**
- **IPC in linux can be implemented using pipe, shared memory, message queue, semaphore, signal or sockets.**

Pipe

- **Pipes are unidirectional byte streams which connect the standard output from one process into the standard input of another process.**
- **A pipe is created using the system call *pipe* that returns a pair of file descriptors.**
- **The descriptor pfd[0] is used for reading and pfd[1] is used for writing.**
- **Can be used only between parent and child processes.**

Shared memory

- **Two or more processes share a single chunk of memory to communicate randomly.**
- **Semaphores are generally used to avoid race condition amongst processes.**
- **Fastest amongst all IPCs as it does not require any system call.**
- **It avoids copying data unnecessarily.**

Message Queue

- **A message queue is a linked list of messages stored within the kernel**
- **A message queue is identified by a unique identifier**
- **Every message has a positive long integer type field, a non-negative length, and the actual data bytes.**
- **The messages need not be fetched on FCFS basis. It could be based on type field.**

Semaphores

- **A semaphore is a counter used to synchronize access to a shared data amongst multiple processes.**
- **To obtain a shared resource, the process should:**
    - **Test the semaphore that controls the resource.**
    - **If value is positive, it gains access and decrements value of semaphore.**
    - **If value is zero, the process goes to sleep and awakes when value is > 0.**
- **When a process relinquishes resource, it increments the value of semaphore by 1.**

Producer-Consumer problem

- **A producer process produces information to be consumed by a consumer process**
- **A producer can produce one item while the consumer is consuming another one.**
- **With bounded-buffer size, consumer must wait if buffer is empty, whereas producer must wait if buffer is full.**
- **The buffer can be implemented using any IPC facility.**

Exp# 5a **Fibonacci & Prime Number**

Aim

To generate 25 fibonacci numbers and determine prime amongst them using pipe.

Algorithm

1. Declare a array to store fibonacci numbers
2. Decalre a array *pfd* with two elements for pipe descriptors.
3. Create pipe on *pfd* using pipe function call.
   a. If return value is -1 then stop
4. Using fork system call, create a child process.
5. Let the child process generate 25 fibonacci numbers and store them in a array.
6. Write the array onto pipe using write system call.
7. Block the parent till child completes using wait system call.
8. Store fibonacci nos. written by child from the pipe in an array using read system call
9. Inspect each element of the fibonacci array and check whether they are prime
   a. If prime then print the fibonacci term.
10. Stop

Result

Thus fibonacci numbers that are prime is determined using IPC pipe.

Program

```
/* Fibonacci and Prime using pipe - fibprime.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

main()
{
   pid_t pid;
   int pfd[2];
   int i,j,flg,f1,f2,f3;
   static unsigned int ar[25],br[25];

   if(pipe(pfd) == -1)
   {
      printf("Error in pipe");
      exit(-1);
   }

   pid=fork();
   if (pid == 0)
   {
      printf("Child process generates Fibonacci series\n" );
      f1 = -1;
      f2 = 1;
      for(i = 0;i < 25; i++)
      {
         f3 = f1 + f2;
         printf("%d\t",f3);
         f1 = f2;
         f2 = f3;
         ar[i] = f3;
      }
      write(pfd[1],ar,25*sizeof(int));
   }
   else if (pid > 0)
   {
      wait(NULL);
      read(pfd[0], br, 25*sizeof(int));
      printf("\nParent prints Fibonacci that are Prime\n");
```

```
      for(i = 0;i < 25; i++)
      {
         flg = 0;
         if (br[i] <= 1)
            flg = 1;
         for(j=2; j<=br[i]/2; j++)
         {
            if (br[i]%j == 0)
            {
               flg=1;
               break;
            }
         }
         if (flg == 0)
            printf("%d\t", br[i]);
      }
      printf("\n");
   }
   else
   {
      printf("Process creation failed");
      exit(-1);
   }
}
```

Output

```
$ gcc fibprime.c

$ ./a.out
Child process generates Fibonacci series
0       1       1       2       3       5       8       13
21      34      55      89      144     233     377     610
987     1597    2584    4181    6765    10946   17711   28657
46368
Parent prints Fibonacci that are Prime
2       3       5       13      89      233     1597    28657
```

Exp# 5b **who | wc -l**

Aim

To determine number of users logged in using pipe.

Algorithm

1. Decalre a array *pfd* with two elements for pipe descriptors.
2. Create pipe on *pfd* using pipe function call.
   a. If return value is -1 then stop
3. Using fork system call, create a child process.
4. Free the standard output (1) using close system call to redirect the output to pipe.
5. Make a copy of write end of the pipe using dup system call.
6. Execute who command using execlp system call.
7. Free the standard input (0) using close system call in the other process.
8. Make a close of read end of the pipe using dup system call.
9. Execute wc –l command using execlp system call.
10. Stop

Result

Thus standard output of who is connected to standard input of wc using pipe to compute number of users logged in.

Program

```
/* No. of users logged - cmdpipe.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
   int pfds[2];
   pipe(pfds);

   if (!fork())
   {
      close(1);
      dup(pfds[1]);
      close(pfds[0]);
      execlp("who", "who", NULL);
   }
   else
   {
      close(0);
      dup(pfds[0]);
      close(pfds[1]);
      execlp("wc", "wc", "-l", NULL);
   }
}
```

Output

```
$ gcc cmdpipe.c

$ ./a.out
15
```

Exp# 5c **Chat Messaging**

Aim

To exchange message between server and client using message queue.

Algorithm

Server

1. Decalre a structure *mesgq* with *type* and *text* fields.
2. Initialize *key* to 2013 (some random value).
3. Create a message queue using msgget with *key* & IPC_CREAT as parameter.
   a. If message queue cannot be created then stop.
4. Initialize the message *type* member of *mesgq* to 1.
5. Do the following until user types Ctrl+D
   a. Get message from the user and store it in *text* member.
   b. Delete the newline character in *text* member.
   c. Place message on the queue using msgsend for the client to read.
   d. Retrieve the response message from the client using msgrcv function
   e. Display the *text* contents.
6. Remove message queue from the system using msgctl with IPC_RMID as parameter.
7. Stop

Client

1. Decalre a structure *mesgq* with *type* and *text* fields.
2. Initialize *key* to 2013 (same value as in server).
3. Open the message queue using msgget with *key* as parameter.
   a. If message queue cannot be opened then stop.
4. Do while the message queue exists
   a. Retrieve the response message from the server using msgrcv function
   b. Display the *text* contents.
   c. Get message from the user and store it in *text* member.
   d. Delete the newline character in *text* member.
   e. Place message on the queue using msgsend for the server to read.
5. Print "Server Disconnected".
6. Stop

Result

Thus chat session between client and server was done using message queue.

Program

Server

```
/* Server chat process - srvmsg.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct mesgq
{
   long type;
   char text[200];
} mq;

main()
{
   int msqid, len;
   key_t key = 2013;

   if((msqid = msgget(key, 0644|IPC_CREAT)) == -1)
   {
      perror("msgget");
      exit(1);
   }

   printf("Enter text, ^D to quit:\n");
   mq.type = 1;

   while(fgets(mq.text, sizeof(mq.text), stdin) != NULL)
   {
      len = strlen(mq.text);
      if (mq.text[len-1] == '\n')
         mq.text[len-1] = '\0';
      msgsnd(msqid, &mq, len+1, 0);

      msgrcv(msqid, &mq, sizeof(mq.text), 0, 0);
      printf("From Client: \"%s\"\n", mq.text);
   }
   msgctl(msqid, IPC_RMID, NULL);
}
```

Client

```
/* Client chat process - climsg.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct mesgq
{
   long type;
   char text[200];
} mq;

main()
{
   int msqid, len;
   key_t key = 2013;

   if ((msqid = msgget(key, 0644)) == -1)
   {
      printf("Server not active\n");
      exit(1);
   }

   printf("Client ready :\n");
   while (msgrcv(msqid, &mq, sizeof(mq.text), 0, 0) != -1)
   {
      printf("From Server: \"%s\"\n", mq.text);

      fgets(mq.text, sizeof(mq.text), stdin);
      len = strlen(mq.text);
      if (mq.text[len-1] == '\n')
         mq.text[len-1] = '\0';
      msgsnd(msqid, &mq, len+1, 0);
   }
   printf("Server Disconnected\n");
}
```

Output

Server

```
$ gcc srvmsg.c -o srvmsg

$ ./srvmsg
Enter text, ^D to quit:
hi
From Client: "hello"
Where r u?
From Client: "I'm where i am"
bye
From Client: "ok"
^D
```

Client

```
$ gcc climsg.c -o climsg

$ ./climsg
Client ready:
From Server: "hi"
hello
From Server: "Where r u?"
I'm where i am
From Server: "bye"
ok
Server Disconnected
```

Exp# 5d **Shared Memory**

Aim

To demonstrate communication between process using shared memory.

Algorithm

Server

1. Initialize size of shared memory *shmsize* to 27.
2. Initialize *key* to 2013 (some random value).
3. Create a shared memory segment using shmget with *key* & IPC_CREAT as parameter.
   a. If shared memory identifier *shmid* is -1, then stop.
4. Display *shmid*.
5. Attach server process to the shared memory using shmmat with *shmid* as parameter.
   a. If pointer to the shared memory is not obtained, then stop.
6. Clear contents of the shared region using memset function.
7. Write a–z onto the shared memory.
8. Wait till client reads the shared memory contents
9. Detatch process from the shared memory using shmdt system call.
10. Remove shared memory from the system using shmctl with IPC_RMID argument
11. Stop

Client

1. Initialize size of shared memory *shmsize* to 27.
2. Initialize *key* to 2013 (same value as in server).
3. Obtain access to the same shared memory segment using same *key*.
   a. If obtained then display the *shmid* else print "Server not started"
4. Attach client process to the shared memory using shmmat with *shmid* as parameter.
   a. If pointer to the shared memory is not obtained, then stop.
5. Read contents of shared memory and print it.
6. After reading, modify the first character of shared memory to '*'
7. Stop

Result

Thus contents written onto shared memory by the server process is read by the client process.

Program

Server

```
/* Shared memory server - shms.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/un.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define shmsize 27

main()
{
   char c;
   int shmid;
   key_t key =  2013;
   char *shm, *s;

   if ((shmid = shmget(key, shmsize, IPC_CREAT|0666)) < 0)
   {
      perror("shmget");
      exit(1);
   }
   printf("Shared memory id : %d\n", shmid);

   if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
   {
      perror("shmat");
      exit(1);
   }

   memset(shm, 0, shmsize);
   s = shm;
   printf("Writing (a-z) onto shared memory\n");
   for (c = 'a'; c <= 'z'; c++)
      *s++ = c;
   *s = '\0';

   while (*shm != '*');
   printf("Client finished reading\n");

   if(shmdt(shm) != 0)
      fprintf(stderr, "Could not close memory segment.\n");

   shmctl(shmid, IPC_RMID, 0);
}
```

Client

```
/* Shared memory client - shmc.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define shmsize 27

main()
{
   int shmid;
   key_t key = 2013;
   char *shm, *s;

   if ((shmid = shmget(key, shmsize, 0666)) < 0)
   {
      printf("Server not started\n");
      exit(1);
   }
   else
      printf("Accessing shared memory id : %d\n",shmid);

   if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
   {
      perror("shmat");
      exit(1);
   }

   printf("Shared memory contents:\n");
   for (s = shm; *s != '\0'; s++)
      putchar(*s);
   putchar('\n');

   *shm = '*';
}
```

Output

Server

```
$ gcc shms.c -o shms

$ ./shms
Shared memory id : 196611
Writing (a-z) onto shared memory
Client finished reading
```

Client

```
$ gcc shmc.c -o shmc

$ ./shmc
Accessing shared memory id : 196611
Shared memory contents:
abcdefghijklmnopqrstuvwxyz
```

Exp# 5e **Producer-Consumer problem**

Aim

To synchronize producer and consumer processes using semaphore.

Algorithm

1. Create a shared memory segment *BUFSIZE* of size 1 and attach it.
2. Obtain semaphore id for variables *empty*, *mutex* and *full* using semget function.
3. Create semaphore for *empty*, *mutex* and *full* as follows:
    a. Declare *semun*, a union of specific commands.
    b. The initial values are: 1 for mutex, N for empty and 0 for full
    c. Use semctl function with SETVAL command
4. Create a child process using fork system call.
    a. Make the parent process to be the *producer*
    b. Make the child process to the *consumer*
5. The *producer* produces 5 items as follows:
    a. Call *wait* operation on semaphores *empty* and *mutex* using semop function.
    b. Gain access to buffer and produce data for consumption
    c. Call *signal* operation on semaphores *mutex* and *full* using semop function.
6. The *consumer* consumes 5 items as follows:
    a. Call *wait* operation on semaphores *full* and *mutex* using semop function.
    b. Gain access to buffer and consume the available data.
    c. Call *signal* operation on semaphores *mutex* and *empty* using semop function.
7. Remove shared memory from the system using shmctl with IPC_RMID argument
8. Stop

Result

Thus synchronization between producer and consumer process for access to a shared memory segment is implemented.

Program

```
/* Producer-Consumer problem using semaphore – pcsem.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define N 5
#define BUFSIZE 1
#define PERMS 0666

int *buffer;
int nextp = 0, nextc = 0;
int mutex, full, empty;    /* semaphore variables */

void producer()
{
   int data;
   if(nextp == N)
      nextp = 0;
   printf("Enter data for producer to produce : ");
   scanf("%d",(buffer + nextp));
   nextp++;
}

void consumer()
{
   int g;
   if(nextc == N)
      nextc = 0;
   g = *(buffer + nextc++);
   printf("\nConsumer consumes data %d", g);
}

void sem_op(int id, int value)
{
   struct sembuf op;
   int v;
   op.sem_num = 0;
   op.sem_op = value;
   op.sem_flg = SEM_UNDO;
   if((v = semop(id, &op, 1)) < 0)
      printf("\nError executing semop instruction");
}
```

```
void sem_create(int semid, int initval)
{
   int semval;
   union semun
   {
      int val;
      struct semid_ds *buf;
      unsigned short *array;
   } s;

   s.val = initval;
   if((semval = semctl(semid, 0, SETVAL, s)) < 0)
      printf("\nError in executing semctl");
}

void sem_wait(int id)
{
   int value = -1;
   sem_op(id, value);
}

void sem_signal(int id)
{
   int value = 1;
   sem_op(id, value);
}

main()
{
   int shmid, i;
   pid_t pid;

   if((shmid = shmget(1000, BUFSIZE, IPC_CREAT|PERMS)) < 0)
   {
      printf("\nUnable to create shared memory");
      return;
   }
   if((buffer = (int*)shmat(shmid, (char*)0, 0)) == (int*)-1)
   {
      printf("\nShared memory allocation error\n");
      exit(1);
   }

   if((mutex = semget(IPC_PRIVATE, 1, PERMS|IPC_CREAT)) == -1)
   {
      printf("\nCan't create mutex semaphore");
      exit(1);
   }
```

```
   if((empty = semget(IPC_PRIVATE, 1, PERMS|IPC_CREAT)) == -1)
   {
      printf("\nCan't create empty semaphore");
      exit(1);
   }
   if((full = semget(IPC_PRIVATE, 1, PERMS|IPC_CREAT)) == -1)
   {
      printf("\nCan't create full semaphore");
      exit(1);
   }

   sem_create(mutex, 1);
   sem_create(empty, N);
   sem_create(full, 0);

   if((pid = fork()) < 0)
   {
      printf("\nError in process creation");
      exit(1);
   }
   else if(pid > 0)
   {
      for(i=0; i<N; i++)
      {
         sem_wait(empty);
         sem_wait(mutex);
         producer();
         sem_signal(mutex);
         sem_signal(full);
      }
   }
   else if(pid == 0)
   {
      for(i=0; i<N; i++)
      {
         sem_wait(full);
         sem_wait(mutex);
         consumer();
         sem_signal(mutex);
         sem_signal(empty);
      }
      printf("\n");
   }
}
```

Output

```
$ gcc pcsem.c

$ ./a.out
Enter data for producer to produce : 5

Enter data for producer to produce : 8
Consumer consumes data 5
Enter data for producer to produce : 4
Consumer consumes data 8
Enter data for producer to produce : 2
Consumer consumes data 4
Enter data for producer to produce : 9
Consumer consumes data 2
Consumer consumes data 9
```

MEMORY MANAGEMENT

- **The first-fit, best-fit, or worst-fit strategy is used to select a free hole from the set of available holes.**

First fit

- **Allocate the first hole that is big enough.**
- **Searching starts from the beginning of set of holes.**

Best fit

- **Allocate the smallest hole that is big enough.**
- **The list of free holes is kept sorted according to size in ascending order.**
- **This strategy produces smallest leftover holes**

Worst fit

- **Allocate the largest hole.**
- **The list of free holes is kept sorted according to size in descending order.**
- **This strategy produces the largest leftover hole.**

The widely used page replacement algorithms are FIFO and LRU.

FIFO

- **Page replacement is based on when the page was brought into memory.**
- **When a page should be replaced, the oldest one is chosen.**
- **Generally, implemented using a FIFO queue.**
- **Simple to implement, but not efficient.**
- **Results in more page faults.**
- **The page-fault may increase, even if frame size is increased (Belady's anomaly)**

LRU

- **Pages used in the recent past are used as an approximation of future usage.**
- **The page that has not been used for a longer period of time is replaced.**
- **LRU is efficient but not optimal.**
- **Implementation of LRU requires hardware support, such as counters/stack.**

Exp# 6a **First Fit Allocation**

Aim

To allocate memory requirements for processes using first fit allocation.

Algorithm

1. Declare structures *hole* and *process* to hold information about set of holes and processes respectively.
2. Get number of holes, say *nh*.
3. Get the size of each hole
4. Get number of processes, say *np*.
5. Get the memory requirements for each process.
6. Allocate processes to holes, by examining each hole as follows:
   a. If hole size > process size then
      i. Mark process as allocated to that hole.
      ii. Decrement hole size by process size.
   b. Otherwise check the next from the set of hole
7. Print the list of process and their allocated holes or unallocated status.
8. Print the list of holes, their actual and current availability.
9. Stop

Result

Thus processes were allocated memory using first fit method.

Program

```
/* First fit allocation - ffit.c */

#include <stdio.h>

struct process
{
   int size;
   int flag;
   int holeid;
} p[10];
struct hole
{
   int size;
   int actual;
} h[10];

main()
{
   int i, np, nh, j;

   printf("Enter the number of Holes : ");
   scanf("%d", &nh);
   for(i=0; i<nh; i++)
   {
      printf("Enter size for hole H%d : ",i);
      scanf("%d", &h[i].size);
      h[i].actual =  h[i].size;
   }

   printf("\nEnter number of process : " );
   scanf("%d",&np);
   for(i=0;i<np;i++)
   {
      printf("enter the size of process P%d : ",i);
      scanf("%d", &p[i].size);
      p[i].flag = 0;
   }
```

```
   for(i=0; i<np; i++)
   {
      for(j=0; j<nh; j++)
      {
         if(p[i].flag != 1)
         {
            if(p[i].size <= h[j].size)
            {
               p[i].flag = 1;
               p[i].holeid = j;
               h[j].size -= p[i].size;
            }
         }
      }
   }

   printf("\n\tFirst fit\n");
   printf("\nProcess\tPSize\tHole");
   for(i=0; i<np; i++)
   {
      if(p[i].flag != 1)
         printf("\nP%d\t%d\tNot allocated", i, p[i].size);
      else
         printf("\nP%d\t%d\tH%d", i, p[i].size, p[i].holeid);
   }

   printf("\n\nHole\tActual\tAvailable");
   for(i=0; i<nh ;i++)
      printf("\nH%d\t%d\t%d", i, h[i].actual, h[i].size);
   printf("\n");
}
```

Output

```
$ gcc ffit.c

$ ./a.out
Enter the number of Holes : 5
Enter size for hole H0 : 100
Enter size for hole H1 : 500
Enter size for hole H2 : 200
Enter size for hole H3 : 300
Enter size for hole H4 : 600

Enter number of process : 4
enter the size of process P0 : 212
enter the size of process P1 : 417
enter the size of process P2 : 112
enter the size of process P3 : 426

        First fit

Process PSize   Hole
P0      212     H1
P1      417     H4
P2      112     H1
P3      426     Not allocated

Hole    Actual  Available
H0      100     100
H1      500     176
H2      200     200
H3      300     300
H4      600     183
```

Exp# 6b **Best Fit Allocation**

Aim

To allocate memory requirements for processes using best fit allocation.

Algorithm

1. Declare structures *hole* and *process* to hold information about set of holes and processes respectively.
2. Get number of holes, say *nh*.
3. Get the size of each hole
4. Get number of processes, say *np*.
5. Get the memory requirements for each process.
6. Allocate processes to holes, by examining each hole as follows:
    a. Sort the holes according to their sizes in ascending order
    b. If hole size > process size then
        i. Mark process as allocated to that hole.
        ii. Decrement hole size by process size.
    c. Otherwise check the next from the set of sorted hole
7. Print the list of process and their allocated holes or unallocated status.
8. Print the list of holes, their actual and current availability.
9. Stop

Result

Thus processes were allocated memory using best fit method.

Program

```
/* Best fit allocation - bfit.c */

#include <stdio.h>

struct process
{
   int size;
   int flag;
   int holeid;
} p[10];
struct hole
{
   int hid;
   int size;
   int actual;
} h[10];

main()
{
   int i, np, nh, j;
   void bsort(struct hole[], int);

   printf("Enter the number of Holes : ");
   scanf("%d", &nh);
   for(i=0; i<nh; i++)
   {
      printf("Enter size for hole H%d : ",i);
      scanf("%d", &h[i].size);
      h[i].actual =  h[i].size;
      h[i].hid = i;
   }

   printf("\nEnter number of process : " );
   scanf("%d",&np);
   for(i=0;i<np;i++)
   {
      printf("enter the size of process P%d : ",i);
      scanf("%d", &p[i].size);
      p[i].flag = 0;
   }

   for(i=0; i<np; i++)
   {
      bsort(h, nh);
```

```
      for(j=0; j<nh; j++)
      {
         if(p[i].flag != 1)
         {
            if(p[i].size <= h[j].size)
            {
               p[i].flag = 1;
               p[i].holeid = h[j].hid;
               h[j].size -= p[i].size;
            }
         }
      }
   }

   printf("\n\tBest fit\n");
   printf("\nProcess\tPSize\tHole");
   for(i=0; i<np; i++)
   {
      if(p[i].flag != 1)
         printf("\nP%d\t%d\tNot allocated", i, p[i].size);
      else
         printf("\nP%d\t%d\tH%d", i, p[i].size, p[i].holeid);
   }

   printf("\n\nHole\tActual\tAvailable");
   for(i=0; i<nh ;i++)
      printf("\nH%d\t%d\t%d", h[i].hid, h[i].actual,
h[i].size);
   printf("\n");
}

void bsort(struct hole bh[], int n)
{
   struct hole temp;
   int i,j;
   for(i=0; i<n-1; i++)
   {
      for(j=i+1; j<n; j++)
      {
         if(bh[i].size > bh[j].size)
         {
            temp = bh[i];
            bh[i] = bh[j];
            bh[j] = temp;
         }
      }
   }
}
```

Output

```
$ gcc bfit.c

$ ./a.out
Enter the number of Holes : 5
Enter size for hole H0 : 100
Enter size for hole H1 : 500
Enter size for hole H2 : 200
Enter size for hole H3 : 300
Enter size for hole H4 : 600

Enter number of process : 4
enter the size of process P0 : 212
enter the size of process P1 : 417
enter the size of process P2 : 112
enter the size of process P3 : 426

        Best fit

Process PSize   Hole
P0      212     H3
P1      417     H1
P2      112     H2
P3      426     H4

Hole    Actual  Available
H1      500     83
H3      300     88
H2      200     88
H0      100     100
H4      600     174
```

Exp# 6c **FIFO Page Replacement**

Aim

To implement demand paging for a reference string using FIFO method.

Algorithm

1. Get length of the reference string, say *l*.
2. Get reference string and store it in an array, say *rs*.
3. Get number of frames, say *nf*.
4. Initalize *frame* array upto length *nf* to -1.
5. Initialize position of the oldest page, say *j* to 0.
6. Initialize no. of page faults, say *count* to 0.
7. For each page in reference string in the given order, examine:
   a. Check whether page exist in the *frame* array
   b. If it does not exist then
      i. Replace page in position *j*.
      ii. Compute page replacement position as (*j*+1) modulus *nf*.
      iii. Increment *count* by 1.
      iv. Display pages in *frame* array.
8. Print *count*.
9. Stop

Result

Thus page replacement was implemented using FIFO algorithm.

Program

```
/* FIFO page replacement - fifopr.c */

#include <stdio.h>

main()
{
   int i,j,l,rs[50],frame[10],nf,k,avail,count=0;

   printf("Enter length of ref. string : ");
   scanf("%d", &l);
   printf("Enter reference string :\n");
   for(i=1; i<=l; i++)
      scanf("%d", &rs[i]);
   printf("Enter number of frames : ");
   scanf("%d", &nf);

   for(i=0; i<nf; i++)
      frame[i] = -1;
   j = 0;

   printf("\nRef. str  Page frames");
   for(i=1; i<=l; i++)
   {
      printf("\n%4d\t", rs[i]);
      avail = 0;
      for(k=0; k<nf; k++)
         if(frame[k] == rs[i])
            avail = 1;
      if(avail == 0)
      {
         frame[j] = rs[i];
         j = (j+1) % nf;
         count++;
         for(k=0; k<nf; k++)
            printf("%4d", frame[k]);
      }
   }
   printf("\n\nTotal no. of page faults : %d\n",count);
}
```

Output

```
$ gcc fifopr.c

$ ./a.out
Enter length of ref. string : 20
Enter reference string :
1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6
Enter number of frames : 5

Ref. str  Page frames
   1        1  -1  -1  -1  -1
   2        1   2  -1  -1  -1
   3        1   2   3  -1  -1
   4        1   2   3   4  -1
   2
   1
   5        1   2   3   4   5
   6        6   2   3   4   5
   2
   1        6   1   3   4   5
   2        6   1   2   4   5
   3        6   1   2   3   5
   7        6   1   2   3   7
   6
   3
   2
   1
   2
   3
   6

Total no. of page faults : 10
```

Exp# 6d **LRU Page Replacement**

Aim

To implement demand paging for a reference string using LRU method.

Algorithm

1. Get length of the reference string, say *len*.
2. Get reference string and store it in an array, say *rs*.
3. Get number of frames, say *nf*.
4. Create *access* array to store counter that indicates a measure of recent usage.
5. Create a function *arrmin* that returns position of minimum of the given array.
6. Initalize *frame* array upto length *nf* to -1.
7. Initialize position of the page replacement, say *j* to 0.
8. Initialize *freq* to 0 to track page frequency
9. Initialize no. of page faults, say *count* to 0.
10. For each page in reference string in the given order, examine:
    a. Check whether page exist in the *frame* array.
    b. If page exist in memory then
        i. Store incremented *freq* for that page position in *access* array.
    c. If page does not exist in memory then
        i. Check for any empty frames.
        ii. If there is an empty frame,
            - Assign that frame to the page
            - Store incremented *freq* for that page position in *access* array.
            - Increment *count*.
        iii. If there is no free frame then
            - Determine page to be replaced using *arrmin* function.
            - Store incremented *freq* for that page position in *access* array.
            - Increment *count*.
        iv. Display pages in *frame* array.
11. Print *count*.
12. Stop

Result

Thus page replacement was implemented using LRU algorithm.

Program

```
/* LRU page replacement - lrupr.c */

#include <stdio.h>

int arrmin(int[], int);

main()
{
   int i,j,len,rs[50],frame[10],nf,k,avail,count=0;
   int access[10], freq=0, dm;

   printf("Length of Reference string : ");
   scanf("%d", &len);
   printf("Enter reference string :\n");
   for(i=1; i<=len; i++)
      scanf("%d", &rs[i]);
   printf("Enter no. of frames : ");
   scanf("%d", &nf);

   for(i=0; i<nf; i++)
      frame[i] = -1;
   j = 0;

   printf("\nRef. str  Page frames");
   for(i=1; i<=len; i++)
   {
      printf("\n%4d\t", rs[i]);
      avail = 0;
      for(k=0; k<nf; k++)
      {
         if(frame[k] == rs[i])
         {
            avail = 1;
            access[k] = ++freq;
            break;
         }
      }
      if(avail == 0)
      {
         dm = 0;
         for(k=0; k<nf; k++)
         {
            if(frame[k] == -1)
               dm = 1;
               break;
         }
```

```
         if(dm == 1)
         {
            frame[k] = rs[i];
            access[k] = ++freq;
            count++;
         }
         else
         {
            j = arrmin(access, nf);
            frame[j] = rs[i];
            access[j] = ++freq;
            count++;
         }
         for(k=0; k<nf; k++)
            printf("%4d", frame[k]);
      }
   }
   printf("\n\nTotal no. of page faults : %d\n", count);
}

int arrmin(int a[], int n)
{
   int i, min = a[0];
   for(i=1; i<n; i++)
      if (min > a[i])
         min = a[i];
   for(i=0; i<n; i++)
      if (min == a[i])
         return i;
}
```

Output

```
$ gcc lrupr.c

$ ./a.out
Length of Reference string : 20
Enter reference string :
1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6
Enter no. of frames : 5

Ref. str  Page frames
   1       1  -1  -1  -1  -1
   2       1   2  -1  -1  -1
   3       1   2   3  -1  -1
   4       1   2   3   4  -1
   2
   1
   5       1   2   3   4   5
   6       1   2   6   4   5
   2
   1
   2
   3       1   2   6   3   5
   7       1   2   6   3   7
   6
   3
   2
   1
   2
   3
   6

Total no. of page faults : 8
```

FILE ALLOCATION

The three methods of allocating disk space are:

1. **Contiguous allocation**
2. **Linked allocation**
3. **Indexed allocation**

Contiguous

- **Each file occupies a set of contiguous block on the disk.**
- **The number of disk seeks required is minimal.**
- **The directory contains address of starting block and number of contiguous block (length) occupied.**
- **Supports both sequential and direct access.**
- **First / best fit is commonly used for selecting a hole.**

Linked

- **Each file is a linked list of disk blocks.**
- **The directory contains a pointer to first and last blocks of the file.**
- **The first block contains a pointer to the second one, second to third and so on.**
- **File size need not be known in advance, as in contiguous allocation.**
- **No external fragmentation.**
- **Supports sequential access only.**

Indexed

- **In indexed allocation, all pointers are put in a single block known as index block.**
- **The directory contains address of the index block.**
- **The i^{th} entry in the index block points to i^{th} block of the file.**
- **Indexed allocation supports direct access.**
- **It suffers from pointer overhead, i.e wastage of space in storing pointers.**

Exp# 7a Contiguous Allocation

Aim

To implement file allocation on free disk space in a contiguous manner.

Algorithm

1. Assume no. of blocks in the disk as 20 and all are free.
2. Display the status of disk blocks before allocation.
3. For each file to be allocated:
   a. Get the *filename*, *start* address and file *length*
   b. If *start* + *length* > 20, then goto step 2.
   c. Check to see whether any block in the range (start, start + length-1) is allocated. If so, then go to step 2.
   d. Allocate blocks to the file contiguously from start block to start + length – 1.
4. Display directory entries.
5. Display status of disk blocks after allocation
6. Stop

Result

Thus contiguous allocation is done for files with the available free blocks.

Program

```
/* Contiguous Allocation - cntalloc.c */

#include <stdio.h>
#include <string.h>

int num=0, length[10], start[10];
char fid[20][4], a[20][4];

void directory()
{
   int i;
   printf("\nFile Start Length\n");
   for(i=0; i<num; i++)
      printf("%-4s %3d %6d\n",fid[i],start[i],length[i]);
}

void display()
{
   int i;
   for(i=0; i<20; i++)
      printf("%4d",i);
   printf("\n");
   for(i=0; i<20; i++)
      printf("%4s", a[i]);
}

main()
{
   int i,n,k,temp,st,nb,ch,flag;
   char id[4];

   for(i=0; i<20; i++)
      strcpy(a[i], "");
   printf("Disk space before allocation:\n");
   display();

   do
   {
      printf("\nEnter File name (max 3 char) : ");
      scanf("%s", id);
      printf("Enter start block : ");
      scanf("%d", &st);
      printf("Enter no. of blocks : ");
      scanf("%d", &nb);
      strcpy(fid[num], id);
      length[num] = nb;
      flag = 0;
```

```
      if((st+nb) > 20)
      {
         printf("Requirement exceeds range\n");
         continue;
      }

      for(i=st; i<(st+nb); i++)
         if(strcmp(a[i], "") != 0)
            flag = 1;
      if(flag == 1)
      {
         printf("Contiguous allocation not possible.\n");
         continue;
      }

      start[num] = st;
      for(i=st; i<(st+nb); i++)
         strcpy(a[i], id);;
      printf("Allocation done\n");
      num++;

      printf("\nAny more allocation (1. yes / 2. no)? : ");
      scanf("%d", &ch);
   } while (ch == 1);

   printf("\n\t\t\tContiguous Allocation\n");
   printf("Directory:");
   directory();
   printf("\nDisk space after allocation:\n");
   display();
   printf("\n");
}
```

Output

```
$ gcc cntalloc.c
$ ./a.out
Disk space before allocation:
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19

Enter File name (max 3 char) : ls
Enter start block : 3
Enter no. of blocks : 4
Allocation done

Any more allocation (1. yes / 2. no)? : 1

Enter File name (max 3 char) : cp
Enter start block : 14
Enter no. of blocks : 3
Allocation done

Any more allocation (1. yes / 2. no)? : 1

Enter File name (max 3 char) : tr
Enter start block : 18
Enter no. of blocks : 3
Requirement exceeds range

Enter File name (max 3 char) : tr
Enter start block : 10
Enter no. of blocks : 3
Allocation done

Any more allocation (1. yes / 2. no)? : 1

Enter File name (max 3 char) : mv
Enter start block : 0
Enter no. of blocks : 2
Allocation done

Any more allocation (1. yes / 2. no)? : 1

Enter File name (max 3 char) : ps
Enter start block : 12
Enter no. of blocks : 3
Contiguous allocation not possible.

Enter File name (max 3 char) : ps
Enter start block : 7
Enter no. of blocks : 3
Allocation done

Any more allocation (1. yes / 2. no)? : 2

                    Contiguous Allocation
Directory:
File Start Length
ls     3      4
cp    14      3
tr    10      3
mv     0      2
ps     7      3

Disk space after allocation:
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
  mv  mv      ls  ls  ls  ls  ps  ps  ps  tr  tr  tr      cp  cp  cp
```